Differential Cost Analysis with Simultaneous Potentials and Anti-potentials

Đorđe Žikelić*
IST Austria
Austria
dzikelic@ist.ac.at

Bor-Yuh Evan Chang†
Amazon
USA
University of Colorado Boulder
USA
byec@amazon.com

Pauline Bolignano
Amazon
UK
pln@amazon.com

Franco Raimondi‡
Amazon
UK
Middlesex University
UK
frai@amazon.com

Abstract

We present a novel approach to differential cost analysis that, given a program revision, attempts to statically bound the difference in resource usage, or cost, between the two program versions. Differential cost analysis is particularly interesting because of the many compelling applications for it, such as detecting resource-use regressions at code-review time or proving the absence of certain side-channel vulnerabilities. One prior approach to differential cost analysis is to apply relational reasoning that conceptually constructs a product program on which one can over-approximate the difference in costs between the two program versions. However, a significant challenge in any relational approach is effectively aligning the program versions to get precise results. In this paper, our key insight is that we can avoid the need for and the limitations of program alignment if, instead, we bound the difference of two cost-bound summaries rather than directly bounding the concrete cost difference. In particular, our method computes a threshold value for the maximal difference in cost between two program versions *simultaneously* using two kinds of cost-bound summaries—a potential function that evaluates to an upper bound for the cost incurred in the first program and an *anti-potential* function that evaluates to a lower bound for the cost incurred in the second. Our method has a number of desirable properties: it can be fully automated, it allows optimizing the threshold value on relative cost, it is suitable for programs that are not syntactically similar, and it supports non-determinism. We have evaluated an implementation of our approach on a number of program pairs collected from the literature, and we find that our method computes tight threshold values on relative cost in most examples.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Software verification**; **Automated static analysis**; **Software performance**.

Keywords: Differential cost analysis, Cost analysis, Relational reasoning, Potential functions

ACM Reference Format:
Đorđe Žikelić, Bor-Yuh Evan Chang, Pauline Bolignano, and Franco Raimondi. 2022. Differential Cost Analysis with Simultaneous Potentials and Anti-potentials. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523435

*This paper describes work performed in part while Đorđe Žikelić was an Applied Scientist Intern at Amazon.
†Bor-Yuh Evan Chang holds concurrent appointments at the University of Colorado Boulder and as an Amazon Scholar. This paper describes work performed at Amazon and is not associated with CU Boulder.
‡Franco Raimondi holds concurrent appointments at Middlesex University and as an Amazon Scholar. This paper describes work performed at Amazon and is not associated with Middlesex.

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523435

1 Introduction

We consider the problem of statically bounding the *difference* in resource usage (i.e., *cost*) between two program versions. In particular, for two program versions and a set of inputs, the goal of differential cost analysis is to compute a *threshold* value t that bounds the maximal difference in cost usage

between the two programs. That is, let numeric-valued variables $\texttt{cost}^{\text{old}}$ and $\texttt{cost}^{\text{new}}$ model the resource usage of the old and new program versions, respectively, then we want to prove the *differential threshold bound* assertion:

$$\texttt{cost}^{\text{new}} - \texttt{cost}^{\text{old}} \leq t$$

on termination of the program versions when given the same input (for all inputs). Notice that our notion of cost is quite generic, and it encompasses metrics such as run time, memory usage, the number of object allocations or the number of thread allocations, etc.

This static analysis problem has many practical applications. For instance, in software development, programs are often modified and extended with new features. A program revision might lead to unacceptable jumps in cost usage. For performance critical software, it is crucial to detect such undesired performance regressions prior to releasing the software to production. With static analysis seeing increasing industrial adoption for verification or bug finding (e.g., in continuous integration pipelines or backing automated code reviews), differential cost analysis is a critical tool to catch potential performance regressions early in development.

One clear way to address the differential cost analysis problem is to apply the *relational approach* [5, 6] where one reasons about the so-called product or relational state of two program versions. There are several recent works on relational cost analysis [14, 16, 43] that do essentially this. These works consider functional programs and use relational type systems for reasoning about and bounding the difference in cost between two programs; the recent work of Qu et al. [42] additionally allows array-manipulating programs with a relational type-and-effect system. While these works present significant advances, a substantial difficulty behind relational reasoning in general is effectively aligning the two programs, as parts that cannot be aligned fall back to less precise unary reasoning.

Our insight in this work is that if we instead take the difference of cost bounds of the two program versions to compare with the differential threshold, then we can sidestep the need for and the limitations of program alignment. Of course, doing this naïvely could be problematic either for soundness or precision. For soundness, we must carefully use both upper and lower bounds on $\texttt{cost}$ (i.e., take the difference between the upper bound of $\texttt{cost}^{\text{new}}$ and the lower bound of $\texttt{cost}^{\text{old}}$). Lower bounds may be a challenge to get, as the existing methods and tools for cost analysis primarily focus on computing upper bounds. For precision, these cost bounds cannot be computed completely independently on the two program versions, as we seek to prove tight threshold bounds when each program version is given the same input.

In this paper, we propose a new method for differential cost analysis in numerical imperative programs with polynomial arithmetic and with non-determinism. Our method uses potential functions from amortized analysis [47] to reason about costs incurred in individual programs. Potential functions are a well-known method for computing upper bounds on the cost incurred in a single program [10, 28–31]. Intuitively, a potential function assigns a "valuation" to a program state that is sufficient to "pay" for the resource use along all paths to a terminating state. What we do in this work is to also use a lower-bound analogue—that we call *anti-potential* functions—to reason about the relative cost between two program versions. An anti-potential function instead assigns a "valuation" to a program state that is *insufficient* to "pay" for the resource use along all paths to a terminating state. While we have drawn inspiration from lower-bound analogues in other domains [21, 39, 48], the key contribution here is computing a differential threshold value on the maximal difference in cost between two programs *simultaneously* with potential and anti-potential functions—one that provides an upper bound on the cost incurred in the new version and the other that provides a lower bound on the cost incurred in the old for the *same inputs*. The simultaneous computation is done by employing a constraint solving-based approach, which collects the necessary constraints on potential and anti-potential functions to serve as upper and lower bounds on incurred cost in the program versions, as well as the differential cost constraint.

The constraint solving-based approach allows our method to provide several key properties: (1) our method can be fully automated, (2) the computation of the threshold value and witnessing potential functions proceed by reduction to linear programming, hence it allows efficient *optimization* of the threshold value by introducing a minimization objective in the linear program, (3) since our method does not depend on syntactic alignment of programs and performs relational reasoning only on the level of inputs, it is suitable for programs that are not syntactically similar, and (4) our method supports non-determinism in the programming language (which can be hard to support with program alignment).

Finally, while we focus on proving differential threshold bound assertions for two program versions and on optimizing the threshold value, we also show that our method can be used to prove any given *symbolic polynomial bound* in terms of program inputs on the difference in cost usage. The reason to focus on concrete threshold values is that they can be optimized, since the real numbers form a well-ordered set. In contrast, it is not clear what would be a correct criterion for optimizing polynomial bounds on a set of inputs, which is the reason why we are only able to prove a given symbolic polynomial bound on the difference in cost usage.

Contributions. Our contributions are as follows:

1. We present a new method for differential cost analysis that uses potential and anti-potential functions to reason about relative incurred cost (Section 4).

2. We give an algorithm for deriving potential and anti-potential functions simultaneously with a threshold value that verifies the differential threshold bound assertion in imperative numerical programs with polynomial arithmetic and non-determinism (Section 5).
3. Our experimental evaluation demonstrates the ability of our method to compute *tight* threshold values for differential cost analysis (Section 6).

Also as a consequence of our approach, we show that the potential and anti-potential method can be adapted to provide a way for computing tight cost bounds on single programs, that is, bounds with precision guarantees (see Section 7).

2 Overview

To illustrate our approach and new concepts, we consider the program pair shown in Fig. 1, which will serve as our running example throughout this paper. We assume that each program has a special variable `cost` that is initialized to 0 and is updated whenever cost is incurred in the program. It is defined by the underlying cost model that could track program run time, memory usage, or any other quantitative property of interest. Cost may take both positive and negative values. In order for the total cost of a program run to be well-defined, we assume that our programs are *terminating*. We elaborate on the need for this assumption in Section 3.

Example 2.1 (An example revision)**.** Fig. 1 shows a revision to a procedure `join` that calls an operator `f` to which a revision was also made. Both versions take two integer arrays `A` and `B` and their lengths `lenA` and `lenB` as inputs, consist of a nested loop that iterates through both arrays, and on each inner loop iteration call the operator `f`. However, the orderings in which the two versions iterate through the arrays differ. Hence, the difference introduced by the revision is reflected in the change of ordering of loops and in the fact that `f` is modified in a way that changes its cost.

Due to the different loop ordering, two versions of `join` cannot be syntactically aligned, making it difficult to apply the relational approach. On the other hand, even the version of this example without the loop interchange shows a pattern that is very easy to imagine in practice, and that may introduce unexpected jumps in cost usage. For instance, suppose that `f` is called through an interface so the programmer that implements `join` may not have the source code of `f` immediately available to them. In such situations, manual detection and quantification of potential increase in cost usage resulting from the revision is not possible, and an automated differential cost analysis is necessary in order to warn the programmer about the potential performance regression. Thus, this example illustrates the challenges in differential cost analysis that our work aims to address.

The differential threshold problem. To quantify the change in cost between two programs, we consider the problem of computing a *threshold* value that bounds the maximal difference in their cost usage when given the same input. We refer to this as the *differential cost analysis problem* (or *DiffCost* for short). A key feature of our approach is that we not only compute a threshold for the DiffCost problem but also *optimize* it to compute as tight of a threshold as possible.

Potential and anti-potential functions. To reason about costs incurred in individual programs, we use potential functions (PFs) [47]. PFs are a well-known method for computing upper bounds on the cost incurred in a single program [10, 28–31]. In this work, we also introduce *anti-potential functions (anti-PFs)*, a notion dual to potential functions that allow us to compute lower bounds on incurred cost.

Informally, a *potential function (PF)* in a program is a function ϕ that assigns a real value to each program state (comprising of a location in the code together with the vector of variable values). It is required to satisfy two conditions, intuitively capturing sufficient resource to reach termination:

Sufficiency preservation. For any reachable state $\mathbf{c}$ in the program and any successor state $\mathbf{c}'$ of $\mathbf{c}$, we have

$$\phi(\mathbf{c}) \geq \phi(\mathbf{c}') + \Delta_{\mathtt{cost}}(\mathbf{c}, \mathbf{c}') \ ,$$

where here we use $\Delta_{\mathtt{cost}}(\mathbf{c}, \mathbf{c}')$ as informal notation for the cost incurred by progressing from $\mathbf{c}$ to $\mathbf{c}'$.

Sufficiency on termination. The ϕ function is nonnegative upon program termination (i.e., $\phi(\mathbf{c}) \geq 0$ for a terminating state $\mathbf{c}$).

Intuitively, these properties impose that ϕ provides enough resources for a program run to progress to a successor state where the resources are used to "pay" for the incurred cost, and that the remaining amount of resources upon termination is nonnegative (i.e., sufficient to "pay" for the execution).

Anti-potential functions (anti-PFs) are required to satisfy dual properties to PFs. An anti-PF χ assigns a real value to each program state and is required to satisfy two conditions, intuitively capturing *in*sufficient resources to reach termination:

Insufficiency preservation. For any reachable state $\mathbf{c}$ and any successor state $\mathbf{c}'$ of $\mathbf{c}$, we have

$$\chi(\mathbf{c}) \leq \chi(\mathbf{c}') + \Delta_{\mathtt{cost}}(\mathbf{c}, \mathbf{c}') \ .$$

Insufficiency on termination. The χ function is nonpositive upon program termination (i.e., $\chi(\mathbf{c}) \leq 0$ for a terminating state $\mathbf{c}$).

Note that these properties for anti-PFs are indeed dual to those imposed by PFs: they require that χ *does not* provide enough resources for a program run to progress from any state to a successor state, and that the amount of resources upon termination is *nonpositive* (i.e., insufficient to "pay" for the execution).

We formally define PFs and anti-PFs in Section 4.1 and then show that for any reachable state $\mathbf{c}$, the values $\phi(\mathbf{c})$ and $\chi(\mathbf{c})$ provide an upper and a lower bound on the cost incurred by any program run that starts in $\mathbf{c}$, respectively.

```
void join(int A[], int lenA, int B[], int lenB) {
  assume(1 ≤ lenA ≤ 100 ∧ 1 ≤ lenB ≤ 100);
  int cost = 0;
ℓ0  χ^old : lenA · lenB
  for (int i = 0;
ℓ1  χ^old : (lenA − i) · lenB
      i < lenA ; i++) {
    for (int j = 0;
ℓ2    χ^old : (lenA − i) · lenB − j
        j < lenB ; j++) {
ℓ3      χ^old : (lenA − i) · lenB − j
      f( A[i] , B[j] , cost );
    }
  }
ℓout  χ^old : 0
}

void f(int a, int b , int &cost ) {
  ...
  cost = cost + 1;
}
```

```
void join(int A[], int lenA, int B[], int lenB) {
  assume(1 ≤ lenA ≤ 100 ∧ 1 ≤ lenB ≤ 100);
  int cost = 0;
  φ^new : 2 · lenB · lenA
  for (int i = 0;
  φ^new : 2 · (lenB − i) · lenA)
      i < lenB ; i++) {
    for (int j = 0;
    φ^new : 2 · ((lenB − i) · lenA − j)
        j < lenA ; j++) {
      φ^new : 2 · ((lenB − i) · lenA − j)
      f( A[j] , B[i] , cost );
    }
  }
  φ^new : 0
}

void f(int a, int b , int &cost ) {
  ...
  cost = cost + 2;
}
```

Figure 1. An example revision of a procedure `join` affecting its cost with the old version on the left and new one on the right. One can see the `join` procedure as representing a common join over two sequences (e.g., arrays, lists, collections via iterators) with an operator `f` having some cost per pair of elements. For subsequent discussion, we label some important program points in `join` with ℓ_0–ℓ_{out} (conceptually, locations in `join`'s control-flow graph). To model the cost of code, it is standard to introduce a `cost` variable, which we can see as ghost state. For improved readability, we highlight the ghost code for updating `cost` in yellow. The goal of differential cost analysis on `join` is to compute a bound on the difference between `cost` in the two versions on termination at program point ℓ_{out}. There are two conceptual changes in this revision. First, in `join` itself, the loops are interchanged, which absent any other changes should not affect the total cost. We highlight deletions in red and additions in green. Second, there is some revision in the `f` operation that changes the cost per pair from 1 to 2 on every input. At program points ℓ_0–ℓ_{out}, we show potentials ϕ and anti-potentials χ that enable proving that the relative cost is at most $\texttt{lenA} \cdot \texttt{lenB}$ (which is bounded by 10,000 when assuming both `lenA` and `lenB` are bounded by 100).

Example 2.2 (Potentials and anti-potentials)**.** Consider again our running example presented in Fig. 1. Examples of a PF ϕ^{new} for the new version of `join` and of an anti-PF χ^{old} for the old version of `join` are presented as annotations in Fig. 1. To see that ϕ^{new} indeed defines a PF for the new version of `join`, observe that a program run incurs cost 2 in each inner loop iteration and otherwise does not incur cost, hence the value of the PF needs to decrease by at least 2 with respect to the transition from ℓ_3 to ℓ_2, and to be non-increasing with respect to other transitions. The expression for ϕ^{new} can be seen to satisfy this property at any reachable state. Note that at any reachable state in the program, we have $\texttt{i} \in [0, \texttt{lenA}]$ and $\texttt{j} \in [0, \texttt{lenB}]$. Moreover, it is nonnegative upon termination. Hence, ϕ^{new} is a PF for the new version of `join`. One analogously verifies that χ^{old} is nonpositive upon termination and that, upon executing each transition, the value of χ^{old} increases at most by the incurred cost. Hence, χ^{old} is an anti-PF in the old version of `join`.

Observe that in both cases, the expression that defines the PF and the anti-PF at each initial state evaluates to the exact cost usage of a program run that starts in that state. Hence, polynomial PFs and anti-PFs allow us to capture the exact cost usage in both versions of `join`.

Using PFs and anti-PFs for differential cost analysis. We now describe how PFs and anti-PFs in combination can be used to reason about the difference in cost between two programs.

Consider the DiffCost problem for two given programs and the set Θ_0 of inputs. Suppose that we are able to compute

a PF ϕ^{new} for the new program and an anti-PF χ^{old} for the old program. Then, in Section 4.2, we prove that for any input $\mathbf{c} \in \Theta_0$, the difference $\phi^{\text{new}}(\mathbf{c}) - \chi^{\text{old}}(\mathbf{c})$ is an *upper bound* on the difference in cost between the two programs on input $\mathbf{c}$. Hence, if a PF ϕ^{new} and an anti-PF χ^{old} satisfy

$$\forall \mathbf{c} \in \Theta_0.\, \phi^{\text{new}}(\mathbf{c}) - \chi^{\text{old}}(\mathbf{c}) \leq t \tag{1}$$

for value t, then t is a threshold for the DiffCost problem. This is the essence of our method for differential cost analysis. In particular, to compute a threshold t for the DiffCost problem, our method computes the following three objects:

1. a PF ϕ^{new} for the new program,
2. an anti-PF χ^{old} for the old program, and
3. a value t which together with ϕ^{new} and χ^{old} satisfies the threshold bound (i.e., eq. (1) above).

In Theorem 4.2 in Section 4.2, we show that this approach to differential cost analysis is not only sound but also theoretically *complete*. In particular, we prove that whenever t is a threshold for the DiffCost problem, there exists a PF ϕ^{new} and an anti-PF χ^{old} for two given programs, which along with the threshold t satisfy eq. (1). We also show in Theorem 4.3, that PFs and anti-PFs can be used to prove that if a given value t is *not* a threshold for the programs, then the difference t in cost can be strictly exceeded on at least one input.

Example 2.3 (Differential cost analysis with potentials and anti-potentials)**.** We now illustrate how this idea can be used to reason about the difference in cost between two programs in Fig. 1. Consider the PF ϕ^{new} and the anti-PF χ^{old} defined in Example 2.2. We have that for each initial program state $\mathbf{c}$, the difference in cost of running the two procedure versions on input $\mathbf{c}$ is at most

$$\phi^{\text{new}}(\mathbf{c}) - \chi^{\text{old}}(\mathbf{c}) = \texttt{lenA} \cdot \texttt{lenB}\ .$$

Since initial variable values are constrained to satisfy $1 \leq \texttt{lenA} \leq 100 \wedge 1 \leq \texttt{lenB} \leq 100$ (as specified by the **assume** statement in Fig. 1), we conclude that

$$\phi^{\text{new}}(\mathbf{c}) - \chi^{\text{old}}(\mathbf{c}) \leq 100 \cdot 100 = 10000$$

holds for any initial state $\mathbf{c}$. Thus, $t = 10000$ is a *threshold* for the DiffCost problem for the revision to the `join` procedure shown in Fig. 1.

Challenge: Deriving PFs and anti-PFs. The idea of using PFs and anti-PFs for differential cost analysis is not entirely surprising once we think of them as a means to obtain upper and lower bounds on incurred cost. However, the key challenge in designing a method for differential cost analysis based on this idea is effectively computing PFs and anti-PFs that would obtain tight threshold values for the DiffCost problem. A naïve approach would be to compute a PF for the new program and an anti-PF for the old program separately, and then to compute a threshold for them. However, such computations of the PF and the anti-PF would not take each other into account, which might lead to imprecision.

Approach: Computing PFs and anti-PFs simultaneously with a threshold. The key conceptual novelty of our method that tackles this challenge is to *simultaneously* compute a threshold t together with a PF ϕ^{new} and an anti-PF χ^{old} that witness it—by employing a constraint solving-based approach. We now outline the key ideas behind our algorithm for computing a threshold t together with a PF ϕ^{new} and an anti-PF χ^{old} that witness it. Further details can be found in Section 5.

For both programs, our method first fixes a polynomial template for a PF by fixing a symbolic polynomial expression over program variables for each location in the program. It also fixes a symbolic variable for the threshold t. Then, the defining properties of PFs and anti-PFs as well as the threshold condition in eq. (1) are all encoded as constraints over the symbolic template variables. This results in a system of constraints, and any solution to the system gives rise to a threshold t as well as to a PF and an anti-PF that witness it.

However, a challenging aspect of constraint solving is that the resulting system of constraints involves both universal quantifiers (e.g., the $\forall \mathbf{c} \in \Theta_0$ in eq. (1)) and polynomial constraints over variables that are hard to solve. Hence, using quantifier elimination to solve these systems of constraints directly would be inefficient. In order to remove universal quantifiers as well as to avoid solving systems of polynomial constraints, our method uses Handelman's theorem [27], a result on positive polynomials from algebraic geometry, to show that these constraints can be converted into *purely existentially quantified* and *linear* constraints over the symbolic template variables. This allows us to reduce the synthesis problem for the threshold and the witnessing PF and anti-PF to solving a system of linear constraints. These linear constraints can then be solved efficiently via an off-the-shelf linear programming (LP) solver. Furthermore, this encoding allows our method to optimize and compute as tight of a threshold as possible, by setting the LP optimization objective to minimize t. In Section 5, we show that our method can also be used to verify a given *symbolic polynomial bound* in terms of program variables on the difference in cost usage, which may be achieved by dropping the minimization objective and replacing the concrete threshold t with the polynomial bound of interest.

Additional Benefit: Precision guarantees on bounds for a single program. While the motivation for this work is to address the differential cost analysis problem, an additional consequence of our approach is that it suggests an approach to compute upper and lower bounds on cost in a single program with *precision guarantees* on the computed bounds. In particular, we show in Section 7 that if we simultaneously compute a PF and an anti-PF for a single program and we regard the threshold t as the maximal difference between the two bounds, then it provides a bound on the *precision* of the computed bounds.

3 Preliminaries

In this work, we consider imperative arithmetic programs with polynomial integer arithmetic. These allow standard programming constructs such as (polynomial) variable assignments, conditional branching and loops. In addition, we allow constructs for *non-deterministic* variable assignments. Cost in programs is modeled by a special program variable `cost`, which is initialized to 0 and modified whenever cost is incurred in the program.

Predicates. Given a finite set of (integer) variables V, a variable valuation is a vector $\mathbf{x} \in \mathbb{Z}^{|V|}$. A *predicate* over V is a set of variable valuations (i.e., a subset of $\mathbb{Z}^{|V|}$). A predicate is said to be a *polynomial assertion* if it is a conjunction of finitely many polynomial inequalities over variables in V. If $\mathbf{x} \in \mathbb{Z}^{|V|}$ and ϕ is a predicate over V given by a boolean formula, we write $\mathbf{x} \models \phi$ to denote that the formula ϕ is satisfied by substituting values of the components of $\mathbf{x}$ for the corresponding variables in ϕ.

Model for programs. We model programs via transition systems. A *transition system* [17] is a tuple $\mathcal{T} = (L, V, \rightarrow, \ell_0, \Theta_0)$, where:

- L is a finite set of *program locations*.
- V is a finite set of *program variables*. We assume that each program has a distinguished variable `cost`.
- $\rightarrow$ is a finite set of *transitions*, which are tuples of the form $\tau = (\ell, \ell', G^\tau, Up^\tau)$. Here, ℓ is the *source* and ℓ' is the *target location* of τ. G^τ is the *guard* of τ, and we assume that it is given by a polynomial assertion over variables V. Finally, Up^τ is the *update* of τ, which to each variable $v \in V$ assigns either a polynomial expression over V with $Up^\tau(v) = v$ if τ does not update v, or the set $\mathbb{Z}$ of integer numbers in case of a non-deterministic variable update.
- ℓ_0 denotes the *initial program location*.
- Θ_0 denotes the set of *initial variable valuations*. We assume that Θ_0 is a polynomial assertion and that for each initial variable valuation, we have $\texttt{cost} = 0$.

We assume the existence of a special *terminal location* ℓ_{out}, which represents the final line of the program code. It has a single outgoing transition $(\ell_{\text{out}}, \ell_{\text{out}}, \text{true}, Up)$ with $Up(v) = v$ for each $v \in V$. Furthermore, we assume that each location $\ell \in L$ has at least one outgoing transition and that it is always possible to execute at least one transition. This is done without loss of generality and may be enforced by introducing a dummy transition from ℓ to ℓ_{out}.

Translation of numerical integer programs into transition systems is standard, so we omit the details. However, for completeness in presentation, we do give the transition systems that model the procedures in Fig. 1 in [50].

A *state* of a transition system is an ordered pair $(\ell, \mathbf{x})$ where $\ell \in L$ and $\mathbf{x} \in \mathbb{Z}^{|V|}$. A state is said to be *initial* if it is of the form $(\ell_0, \mathbf{x}_0)$ with $\mathbf{x}_0 \in \Theta_0$. A state is said to be *terminal* if it is of the form $(\ell_{\text{out}}, \mathbf{x})$. A state $(\ell', \mathbf{x}')$ is a *successor* of a state $(\ell, \mathbf{x})$ if there exists a transition $\tau = (\ell, \ell', G^\tau, Up^\tau)$ with $\mathbf{x} \models G^\tau$ and $Up^\tau(v)(\mathbf{x}) = \mathbf{x}'[v]$ for each $v \in V$ whose update is deterministic. Given a state $\mathbf{c}$, a *finite path from* $\mathbf{c}$ in $\mathcal{T}$ is a finite sequence of states $\mathbf{c}_0 = \mathbf{c}, \mathbf{c}_1, \ldots, \mathbf{c}_k$, where for each $0 \leq i < k$ we have that $\mathbf{c}_{i+1}$ is a successor of $\mathbf{c}$. A state $\mathbf{c}'$ is *reachable from* $\mathbf{c}$ if there exists a finite path from $\mathbf{c}$ that ends in $\mathbf{c}'$. A *run (or execution) from* $\mathbf{c}$ is an infinite sequence of program states where each finite prefix is a finite path from $\mathbf{c}$. When we omit specifying a state $\mathbf{c}$, we refer to a finite path, run or reachability from some initial state.

For a state $\mathbf{c}$, we define $Run(\mathbf{c})$ to be the set of all runs in $\mathcal{T}$ that start in $\mathbf{c}$. We denote by $Run_{\mathcal{T}}$ to be the set of all runs in $\mathcal{T}$ that start in some initial state of $\mathcal{T}$.

Cost of a run and the termination assumption. In order for the notion of the cost of a run to be well-defined, in the rest of this work we assume that all our transition systems (and thus programs) are terminating. Given a transition system $\mathcal{T}$ and a state $\mathbf{c}$, we say that a run from $\mathbf{c}$ is *terminating* if it has a finite prefix with the last state being terminal. We then say that the transition system $\mathcal{T}$ is *terminating* if every run in $\mathcal{T}$ from some initial state is terminating.

Given a run ρ from some state $(\ell, \mathbf{x})$, let $(\ell_{\text{out}}, \mathbf{x}_{out})$ be the terminal state reached in ρ. Then the *cost* of ρ, denoted by $Cost_{\mathcal{T}}(\rho)$, equals the difference of the terminal and the initial value of the variable `cost` along ρ (i.e., $\mathbf{x}_{out}[\texttt{cost}] - \mathbf{x}[\texttt{cost}]$). We define the *maximal cost* of $(\ell, \mathbf{x})$ to be the maximal cost incurred by any run in $\mathcal{T}$ from $(\ell, \mathbf{x})$, that is,

$$CostSup_{\mathcal{T}}(\ell, \mathbf{x}) = \sup \left\{ Cost_{\mathcal{T}}(\rho) \mid \rho \in Run(\ell, \mathbf{x}) \right\} .$$

Similarly, we define the *minimal cost* of $(\ell, \mathbf{x})$ to be

$$CostInf_{\mathcal{T}}(\ell, \mathbf{x}) = \inf \left\{ Cost_{\mathcal{T}}(\rho) \mid \rho \in Run(\ell, \mathbf{x}) \right\} .$$

Note that these two values might differ due to the possible existence of non-determinism in the program. However, we always have $CostInf_{\mathcal{T}}(\ell, \mathbf{x}) \leq CostSup_{\mathcal{T}}(\ell, \mathbf{x})$.

The termination assumption is essential for the cost of a run to be well-defined. Indeed, for a non-terminating run one can naively try to define its cost by taking the limit of costs of its finite prefixes. However, this limit does not need to necessarily exist as we allow both positive and negative costs. Allowing costs of arbitrary sign is necessary to model some of the most important use cases of cost analysis. For instance, in order to obtain tight bounds on memory usage in programs, we need to take into account that memory can be released and returned to the program which is modeled by incurring negative cost.

The differential cost analysis problem. We now formally define the differential cost analysis problem that we consider in this work. Given two programs, our goal is to compare their cost usage and to compute a bound on the maximal difference in incurred cost between the two programs given the same input.

Formally, let $\mathcal{T}^{\text{new}} = (L^{\text{new}}, V, \to^{\text{new}}, \ell_0^{\text{new}}, \Theta_0)$ and $\mathcal{T}^{\text{old}} = (L^{\text{old}}, V, \to^{\text{old}}, \ell_0^{\text{old}}, \Theta_0)$ be two transition systems that share the same finite set of variables V and the set Θ_0 of initial variable valuations. The *differential cost analysis (DiffCost) problem* asks to compute a *threshold* $t \in \mathbb{Z}$ on the difference in cost usage that cannot be exceeded, that is, $t \in \mathbb{Z}$ for which the following logical formula is true:

$$\forall \mathbf{x} \in \Theta_0.\ \mathit{CostSup}_{\mathcal{T}^{\text{new}}}(\ell_0^{\text{new}}, \mathbf{x}) - \mathit{CostInf}_{\mathcal{T}^{\text{old}}}(\ell_0^{\text{old}}, \mathbf{x}) \le t.$$

4 Potential and Anti-potential Functions for Differential Cost Analysis

Potential functions (PFs) from amortized analysis [47] are a classical method for computing upper bounds on the cost incurred in a single program. In this work we show that PFs and their dual for computing lower bounds on incurred cost, which we call *anti-potential functions (anti-PFs)*, can also be used to reason about differential cost analysis.

4.1 Potential and Anti-potential Functions

Let $\mathcal{T}$ be a transition system. A *potential function (PF)* in $\mathcal{T}$ is a map ϕ that assigns a real value to each state in $\mathcal{T}$, and which satisfies the following two properties:

Sufficiency preservation. For any reachable state $(\ell, \mathbf{x})$ in $\mathcal{T}$ and any successor state $(\ell', \mathbf{x}')$ of $(\ell, \mathbf{x})$, we have

$$\phi(\ell, \mathbf{x}) \ge \phi(\ell', \mathbf{x}') + \mathbf{x}'[\texttt{cost}] - \mathbf{x}[\texttt{cost}].$$

We use $\mathbf{x}[\texttt{cost}]$ and $\mathbf{x}'[\texttt{cost}]$ to denote the values of the variable `cost` defined by valuations $\mathbf{x}$ and $\mathbf{x}'$.

Sufficiency on termination. For any reachable terminal state $(\ell_{\text{out}}, \mathbf{x})$ in $\mathcal{T}$, we have $\phi(\ell_{\text{out}}, \mathbf{x}) \ge 0$.

One can define a notion dual to PFs in order to compute lower bounds on the cost usage of a given program, provided that the program is terminating. An *anti-potential function (anti-PF)* in $\mathcal{T}$ is a map χ, which to each state in $\mathcal{T}$, assigns a real value, and which satisfies the following two properties:

Insufficiency preservation. For any reachable state $(\ell, \mathbf{x})$ in $\mathcal{T}$ and any successor state $(\ell', \mathbf{x}')$ of $(\ell, \mathbf{x})$, we have

$$\chi(\ell, \mathbf{x}) \le \chi(\ell', \mathbf{x}') + \mathbf{x}'[\texttt{cost}] - \mathbf{x}[\texttt{cost}].$$

Insufficiency on termination. For any reachable terminal state $(\ell_{\text{out}}, \mathbf{x})$ in $\mathcal{T}$, we have $\chi(\ell_{\text{out}}, \mathbf{x}) \le 0$.

The following theorem shows that, given a reachable state $(\ell, \mathbf{x})$ in $\mathcal{T}$, PFs and anti-PFs evaluate to upper bounds on the maximal cost and lower bounds on the minimal cost of a run starting $(\ell, \mathbf{x})$, respectively. Observe that, in order to prove these inequalities, it suffices to prove that for any run ρ starting in $(\ell, \mathbf{x})$, the values $\phi(\ell, \mathbf{x})$ and $\chi(\ell, \mathbf{x})$ are respectively an upper and a lower bound on the cost of ρ. We prove this by induction on the length of ρ, and the proof can be found in the extended version of the paper [50].

Theorem 4.1. *Let $\mathcal{T}$ be a transition system that is terminating. If ϕ is a PF in $\mathcal{T}$, then for any reachable state $(\ell, \mathbf{x})$ in $\mathcal{T}$, we have*

$$\phi(\ell, \mathbf{x}) \ge \mathit{CostSup}_{\mathcal{T}}(\ell, \mathbf{x}).$$

If χ is an anti-PF in $\mathcal{T}$, then for any reachable state $(\ell, \mathbf{x})$ in $\mathcal{T}$ we have

$$\chi(\ell, \mathbf{x}) \le \mathit{CostInf}_{\mathcal{T}}(\ell, \mathbf{x}).$$

Recall, Example 2.2 in Section 2 shows a pair of a PF and an anti-PF in two versions of the procedure `join` in Fig. 1.

We note that the termination assumption is necessary for Theorem 4.1 to hold, as the claim for anti-PFs may be violated even if all incurred costs are of the same sign so that the cost of a run in a non-terminating program is well-defined. An example program demonstrating this necessity is provided in the extended version of the paper [50].

4.2 Application to Differential Cost Analysis

We now proceed to show how PFs and anti-PFs can be used to reason about differential cost analysis. Let $\mathcal{T}^{\text{new}} = (L^{\text{new}}, V, \to^{\text{new}}, \ell_0^{\text{new}}, \Theta_0)$ and $\mathcal{T}^{\text{old}} = (L^{\text{old}}, V, \to^{\text{old}}, \ell_0^{\text{old}}, \Theta_0)$ be two terminating transition systems that share the finite set of variables V and the set Θ_0 of initial variable valuations. The following theorem shows that PFs and anti-PFs are *sound* for computing the threshold on the maximal difference in cost usage for the DiffCost problem that we defined in Section 3. It also shows that they are theoretically *complete*, in the sense that any valid threshold for the DiffCost problem admits a pair of a PF in $\mathcal{T}^{\text{new}}$ and an anti-PF in $\mathcal{T}^{\text{old}}$ that witness it.

The proof of soundness follows from Theorem 4.1, and the proof of completeness follows by observing that the maximal and minimal costs for each state satisfy the defining properties of PFs and anti-PFs. The proof can be found in [50].

Theorem 4.2 (PFs and anti-PFs for DiffCost)**.** *Let $\mathcal{T}^{\text{new}}$ and $\mathcal{T}^{\text{old}}$ be two terminating transition systems. Suppose that ϕ^{new} is a PF in $\mathcal{T}^{\text{new}}$ and that χ^{old} is an anti-PF in $\mathcal{T}^{\text{old}}$. Then, for each initial variable valuation $\mathbf{x} \in \Theta_0$, we have that*

$$\begin{aligned}&\mathit{CostSup}_{\mathcal{T}^{\text{new}}}(\ell_0^{\text{new}}, \mathbf{x}) - \mathit{CostInf}_{\mathcal{T}^{\text{old}}}(\ell_0^{\text{old}}, \mathbf{x})\\ &\le \phi^{\text{new}}(\ell_0^{\text{new}}, \mathbf{x}) - \chi^{\text{old}}(\ell_0^{\text{old}}, \mathbf{x}).\end{aligned}$$

In particular, if t satisfies $\phi^{\text{new}}(\ell_0^{\text{new}}, \mathbf{x}) - \chi^{\text{old}}(\ell_0^{\text{old}}, \mathbf{x}) \le t$ for each $\mathbf{x} \in \Theta_0$, then t is a threshold for the DiffCost problem.

Conversely, if $t \in \mathbb{Z}$ is a threshold for the DiffCost problem, then there exist a PF ϕ^{new} in $\mathcal{T}^{\text{new}}$ and an anti-PF χ^{old} in $\mathcal{T}^{\text{old}}$ such that $\phi^{\text{new}}(\ell_0^{\text{new}}, \mathbf{x}) - \chi^{\text{old}}(\ell_0^{\text{old}}, \mathbf{x}) \le t$ holds for each $\mathbf{x} \in \Theta_0$.

Hence, in order to compute a threshold t for the DiffCost problem, it suffices to compute the following three objects:

1. a PF ϕ^{new} for $\mathcal{T}^{\text{new}}$,
2. a anti-PF χ^{old} for $\mathcal{T}^{\text{old}}$, and
3. an integer $t \in \mathbb{Z}$ which together with ϕ^{new} and χ^{old} satisfies $\phi^{\text{new}}(\ell_0^{\text{new}}, \mathbf{x}) - \chi^{\text{old}}(\ell_0^{\text{old}}, \mathbf{x}) \le t$ for each initial variable valuation $\mathbf{x} \in \Theta_0$.

We already illustrated this idea in Example 2.3, Section 2, on our running example in Fig. 1.

4.3 Refuting a Threshold via PFs and Anti-PFs

We conclude this section by showing that PFs and anti-PFs can also be used to prove that some threshold in differential cost can be *strictly exceeded*. The proof can be found in the extended version [50]. While the approach for refuting a threshold value is sound for general programs with non-determinism, it is complete only for deterministic programs.

Theorem 4.3 (Refuting a threshold)**.** *Let $\mathcal{T}^{\text{new}}$ and $\mathcal{T}^{\text{old}}$ be two terminating transition systems. Suppose that χ^{new} is an anti-PF in $\mathcal{T}^{\text{new}}$, and that ϕ^{old} is a PF in $\mathcal{T}^{\text{old}}$. Then, for each initial variable valuation $\mathbf{x} \in \Theta_0$, we have that*

$$\begin{aligned} &\mathit{CostInf}_{\mathcal{T}^{\text{new}}}(\ell_0^{\text{new}}, \mathbf{x}) - \mathit{CostSup}_{\mathcal{T}^{\text{old}}}(\ell_0^{\text{old}}, \mathbf{x}) \\ &\geq \chi^{\text{new}}(\ell_0^{\text{new}}, \mathbf{x}) - \phi^{\text{old}}(\ell_0^{\text{old}}, \mathbf{x}). \end{aligned}$$

In particular, if $t \in \mathbb{Z}$ satisfies $\chi^{\text{new}}(\ell_0^{\text{new}}, \mathbf{x}) - \phi^{\text{old}}(\ell_0^{\text{old}}, \mathbf{x}) > t$ for some $\mathbf{x} \in \Theta_0$, then t is not *a threshold for the DiffCost problem.*

Conversely, if t is not a threshold for the DiffCost problem and if $\mathcal{T}^{\text{new}}$ and $\mathcal{T}^{\text{old}}$ are induced by deterministic programs, then there exist an anti-PF χ^{new} in $\mathcal{T}^{\text{new}}$ and a PF ϕ^{old} in $\mathcal{T}^{\text{old}}$ such that $\chi^{\text{new}}(\ell_0^{\text{new}}, \mathbf{x}) - \phi^{\text{old}}(\ell_0^{\text{old}}, \mathbf{x}) > t$ for at least one $\mathbf{x} \in \Theta_0$.

Example 4.4. To illustrate how PFs and anti-PFs can be used to prove that some threshold in cost difference can be exceeded, consider again our running example in Fig. 1, and the PF ϕ^{new} and the anti-PF χ^{old} that were defined in Example 2.2. By analogous reasoning as in Example 2.2, one may easily check that $\chi^{\text{new}} = \phi^{\text{new}}$ is also an anti-PF in the new version of `join` and that $\phi^{\text{old}} = \chi^{\text{old}}$ is a PF in the old version of `join`. On the other hand, from the **assume** statements in Fig. 1 we see that the initial variable valuations in both programs are given by the assertion $\Theta_0 = 1 \leq \texttt{lenA} \leq 100 \wedge 1 \leq \texttt{lenB} \leq 100$. Hence, for any initial variable valuation $\mathbf{x} \in \Theta_0$, we have

$$\begin{aligned} &\mathit{CostInf}_{\mathcal{T}^{\text{new}}}(\ell_0^{\text{new}}, \mathbf{x}) - \mathit{CostSup}_{\mathcal{T}^{\text{old}}}(\ell_0^{\text{old}}, \mathbf{x}) \\ &\geq \chi^{\text{new}}(\ell_0^{\text{new}}, \mathbf{x}) - \phi^{\text{old}}(\ell_0^{\text{old}}, \mathbf{x}) = \texttt{lenA} \cdot \texttt{lenB}. \end{aligned}$$

Thus, as $\texttt{lenA} \cdot \texttt{lenB} = 10000$ for any initial variable valuation in Θ_0 with $\texttt{lenA} = 100$ and $\texttt{lenB} = 100$, it follows that, $t = 9999$ is not a threshold for the DiffCost problem for our running example in Fig. 1.

We elaborate on the reason why PFs and anti-PFs are complete for refuting threshold values only in deterministic programs. Suppose that $\chi^{\text{new}}(\ell_0^{\text{new}}, \mathbf{x}) - \phi^{\text{old}}(\ell_0^{\text{old}}, \mathbf{x}) > t$ holds for some $\mathbf{x} \in \Theta_0$. Then, for *every run* ρ^{new} starting in $(\ell_0^{\text{new}}, \mathbf{x})$ in $\mathcal{T}^{\text{new}}$ and *every run* ρ^{old} starting in $(\ell_0^{\text{old}}, \mathbf{x})$ in $\mathcal{T}^{\text{old}}$, the cost of ρ^{new} exceeds the cost of ρ^{old} by an amount that is strictly greater than t. However, for t not to be a threshold value, it would suffice that this happens for a single pair of such runs. PFs and anti-PFs cannot be used to witness this weaker condition. In the case of deterministic programs, however, there is only a single run ρ^{new} starting in $(\ell_0^{\text{new}}, \mathbf{x})$ and a single run ρ^{old} starting in $(\ell_0^{\text{old}}, \mathbf{x})$, hence PFs and anti-PFs are complete for refuting threshold values in deterministic programs.

5 Simultaneous Potentials and Anti-potentials Algorithm

We now present our algorithm for differential cost analysis. Our algorithm is based on the idea that was presented in Section 4.2, and it simultaneously computes a polynomial PF for the new program, a polynomial anti-PF for the old program, as well as a threshold value. The algorithm runs in polynomial time, and reduces the computation of the PF, the anti-PF and the threshold value to a linear programming (LP) instance by employing a constraint solving-based approach.

In what follows, let $\mathcal{T}^{\text{new}} = (L^{\text{new}}, V, \rightarrow^{\text{new}}, \ell_0^{\text{new}}, \Theta_0)$ and $\mathcal{T}^{\text{old}} = (L^{\text{old}}, V, \rightarrow^{\text{old}}, \ell_0^{\text{old}}, \Theta_0)$ be two terminating transition systems that share the finite set of variables V and the set Θ_0 of initial variable valuations.

Algorithm assumptions. Our algorithm has two constant natural number parameters $d, K \in \mathbb{N}$, where d is the maximal degree of polynomials that it considers and K is a parameter whose meaning we will explain shortly. It also assumes the following:

1. *Affine invariants.* Recall that the defining properties of PFs and anti-PFs impose conditions on their values at *reachable* states in programs. In order to compute PFs and anti-PFs, our algorithm assumes that it is provided with *invariants* I^{new} for $\mathcal{T}^{\text{new}}$ and I^{old} for $\mathcal{T}^{\text{old}}$. An invariant is an over-approximation of the set of all reachable program states. Formally, an invariant in a program is a map I that to each program location ℓ assigns a predicate $I(\ell)$, such that for any reachable state $(\ell, \mathbf{x})$ in the program we have $\mathbf{x} \models I(\ell)$. We assume that the pre-computed invariants are *affine*, meaning that each $I(\ell)$ is given by a conjunction of finitely many affine inequalities over program variables. Affine invariant generation is a well-studied problem in program analysis, with several efficient methods and tools [20, 44].
2. Each transition guard is assumed to be given in terms of affine inequalities. This assumption is made without loss of generality, as any non-affine expression may be replaced by a dummy variable to which the value of this expression is previously assigned.
3. Θ_0 is assumed to be a conjunction of affine inequalities.

Constraint solving-based approach. In order to compute a threshold value for the DiffCost problem, our algorithm employs a constraint solving-based approach to simultaneously search for a polynomial PF ϕ^{new} in $\mathcal{T}^{\text{new}}$, a polynomial anti-PF χ^{old} in $\mathcal{T}^{\text{old}}$, and a threshold value t; it proceeds in 4 steps. First, the algorithm fixes symbolic polynomial templates for ϕ^{new} and χ^{old}, as well as a symbolic template variable for t. Second, the algorithm collects the defining properties of PFs, anti-PFs and the differential cost constraint. Third, the collected constraints are soundly converted into a system of purely existentially quantified linear constraints. Fourth, the

resulting system of linear constraints is efficiently solved by an off-the-shelf LP solver. The threshold value is in addition minimized, in order to compute a bound as tight as possible on relative cost. In what follows, we describe each of these 4 steps in more detail.

Step 1: Symbolic templates. The algorithm fixes a symbolic polynomial template of degree at most d for ϕ^{new} by introducing symbolic polynomial $\phi^{\text{new}}(\ell)$ of degree at most d for each location $\ell \in L^{\text{new}}$. This is done as follows. Let $\text{Mono}_d(V)$ be the set of all monomials of degree at most d over the variable set V. Then, the symbolic template for $\phi^{\text{new}}(\ell)$ is a symbolic polynomial expression $\sum_{f \in \text{Mono}_d(V)} u_f^\ell \cdot f$, where u_f^ℓ is a real-valued symbolic template variable for each $f \in \text{Mono}_d(V)$.

Similarly, the algorithm fixes a symbolic polynomial template of degree at most d for χ^{old}. Finally, it fixes a real-valued symbolic template variable t for the threshold value.

Step 2: Constraint collection. The algorithm now collects all the defining constraints for ϕ^{new} to be a PF in $\mathcal{T}^{\text{new}}$, for χ^{old} to be an anti-PF in $\mathcal{T}^{\text{old}}$ and for t to be a threshold value:

1. *PF constraints.* Recall that a PF needs to satisfy the *sufficiency* preservation condition at all reachable states and the sufficiency on termination condition at all reachable terminal states. To capture that a state is reachable, our algorithm collects constraints which impose the defining conditions at all states contained in the invariant I^{new}. In particular, our algorithm collects the following constraints:
   - For each transition $\tau = (\ell, \ell', G^\tau, Up^\tau) \in \to^{\text{new}}$,
   $$\mathbf{x} \models I^{\text{new}}(\ell) \cap G^\tau \Rightarrow \phi^{\text{new}}(\ell, \mathbf{x}) \geq \phi^{\text{new}}(\ell', Up^\tau(\mathbf{x})) + Up^\tau(\texttt{cost})(\mathbf{x}) - \mathbf{x}[\texttt{cost}].$$
     Here, $\phi^{\text{new}}(\ell', Up^\tau(\mathbf{x}))$ is a notation for the expression obtained by taking the template polynomial for ϕ^{new} at ℓ' and substituting for each variable $v \in V$ either the polynomial update $Up^\tau(v)(\mathbf{x})$, or a fresh variable in the case of a non-deterministic update.
   - $\mathbf{x} \models I^{\text{new}}(\ell_{\text{out}}) \Rightarrow \phi^{\text{new}}(\ell_{\text{out}}, \mathbf{x}) \geq 0$.
2. *Anti-PF constraints.* Similarly, an anti-PF needs to satisfy the *insufficiency* preservation and the insufficiency on termination conditions, so our algorithm collects the following constraints:
   - For each transition $\tau = (\ell, \ell', G^\tau, Up^\tau) \in \to^{\text{old}}$,
   $$\mathbf{x} \models I^{\text{old}}(\ell) \cap G^\tau \Rightarrow \chi^{\text{old}}(\ell, \mathbf{x}) \leq \chi^{\text{old}}(\ell', Up^\tau(\mathbf{x})) + Up^\tau(\texttt{cost})(\mathbf{x}) - \mathbf{x}[\texttt{cost}].$$
     where $\chi^{\text{old}}(\ell', Up^\tau(\mathbf{x}))$ is analogous to $\phi^{\text{new}}(\ell', Up^\tau(\mathbf{x}))$.
   - $\mathbf{x} \models I^{\text{old}}(\ell_{\text{out}}) \Rightarrow \chi^{\text{old}}(\ell_{\text{out}}, \mathbf{x}) \leq 0$.
3. *Differential cost constraint.* Our algorithm then collects the differential cost constraint:
   $$\mathbf{x} \models \Theta_0 \Rightarrow \phi^{\text{new}}(\ell_0^{\text{new}}, \mathbf{x}) - \chi^{\text{old}}(\ell_0^{\text{old}}, \mathbf{x}) \leq t.$$

Observe that for each collected constraint, the expressions on the right-hand–side are linear in symbolic template variables for ϕ^{new}, χ^{old} and t, and the expressions on the left-hand–side contain no symbolic template variables.

Step 3: Conversion to a linear program. The algorithm now converts each collected constraint into a system of purely existentially quantified linear constraints. To do this, we observe that each collected constraint has the following form:

$$\text{aff}_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge \text{aff}_k(\mathbf{x}) \geq 0 \Rightarrow \text{poly}(\mathbf{x}) \geq 0, \quad (2)$$

for some $k \geq 0$, where each aff_i is an affine expression and poly is a polynomial expression over program variables. This is because the left-hand-side of each constraint collected in Step 2 depends either on program invariants, transition guards or the initial variable valuation set Θ_0, all of which are assumed to be affine.

The algorithm converts the constraint in eq. (2) into a system of linear constraints by requiring poly to be equal to a non-negative linear combination of finitely many products of affine expressions in $\text{Aff} = \{\text{aff}_1, \ldots, \text{aff}_k\}$. To formalize this, we define $\text{Prod}_K(\text{Aff})$ to be the set of products of at most K affine expressions in Aff, i.e.

$$\text{Prod}_K(\text{Aff}) = \left\{ \prod_{i=1}^{t} a_i \;\middle|\; t \in \mathbb{N}_0,\ t \leq d,\ a_1, \ldots, a_t \in \text{Aff} \right\}.$$

Recall that K is one of the algorithm's two natural number parameters, and we introduce it in order to bound the maximal number of affine expressions that may appear in each product. One can see that any $g \in \text{Prod}_K(\text{Aff})$ satisfies $g(\mathbf{x}) \geq 0$ for any valuation $\mathbf{x}$ in which $\text{aff}_i(\mathbf{x}) \geq 0$ for each $\text{aff}_i \in \text{Aff}$. Hence, one can soundly translate the constraint in eq. (2) into another constraint that encodes that poly can be written as a non-negative linear combination of finitely many products of affine expressions in Aff:

$$\text{poly}(\mathbf{x}) = \sum_{g \in \text{Prod}_K(\text{Aff})} c_g \cdot g(\mathbf{x}), \quad (3)$$

where each $c_g \geq 0$.

To encode eq. (3) as a system of linear constraints, our algorithm introduces a fresh symbolic variable c_g and a constraint $c_g \geq 0$ for each $g \in \text{Prod}_K(\text{Aff})$. Then, for each monomial over the variable set V it equalizes the coefficients of the monomial on two sides of eq. (3) to produce a linear constraint over the symbolic template variables for ϕ^{new}, χ^{old} and t, as well as the symbolic variables c_g. Collecting all the produced constraints results in a sound translation of the constraint in eq. (2) into a system of purely existentially quantified linear constraints over symbolic variables.

While the translation of eq. (2) into eq. (3) is sound, it is not necessarily complete. However, Handelman's theorem implies that, if we work over real arithmetic and if the set $\langle \text{Aff} \rangle = \{\mathbf{x} \in \mathbb{R}^V \mid \text{aff}_i(\mathbf{x}) \geq 0 \text{ for each } \text{aff}_i \in \text{Aff}\}$ is topologically compact (i.e., closed and bounded), satisfiability of eq. (3) is also a necessary condition for poly to be strictly positive over $\langle \text{Aff} \rangle$.

Handelman's Theorem [27]. *Let V be a finite set of real variables and $Aff = \{aff_1, \ldots, aff_k\}$ be a set of finitely many affine expressions over V (degree 1 polynomials). Let $f \in \mathbb{R}[V]$ be a polynomial and suppose that $f(\mathbf{x}) > 0$ for all $\mathbf{x} \in \langle Aff \rangle = \{\mathbf{x} \in \mathbb{R}^V \mid aff_i(\mathbf{x}) \geq 0 \text{ for each } aff_i \in Aff\}$. If $\langle Aff \rangle$ is a compact set, then there exist $d \in \mathbb{N}_0$ and $s \in \mathbb{N}_0$ such that*

$$f = \sum_{i=1}^{s} c_i \cdot g_i$$

for some $c_1, \ldots, c_s \geq 0$ and $g_1, \ldots, g_s \in Prod_K(\Gamma)$.

Hence, if $\langle \text{Aff} \rangle$ is a compact set, the translation of eq. (2) into eq. (3) is also a complete method to impose a slightly different constraint than the one in eq. (2), with the strict inequality $>$ instead of $\geq$ on the right-hand–side and the inequalities on the left-hand–side being imposed for all real-valued valuations $\mathbf{x}$ (and not just integer-valued). Note that $\langle \text{Aff} \rangle$ being compact is an assumption that is satisfied whenever all variable values in programs are bounded, including the variable `cost` that tracks the total resource usage. Indeed, if we know variable value bounds then we may add these bounds to the invariant at each program location as well as to Θ_0. Since the set Aff induced by any constraint collected in Step 2 contains either inequalities that define an invariant at some program location or inequalities that define Θ_0, this modification would result in Aff being compact.

Step 4: Synthesis via constraint solving. Denote by Φ the set of all linear constraints produced in Step 3. The algorithm employs an off-the-shelf LP solver to solve

$$\begin{aligned} &\text{minimize } t \\ &\text{subject to } \Phi\ . \end{aligned}$$

The algorithm then outputs the computed t if the LP solver finds a solution, or "Unknown" otherwise.

The following theorem establishes soundness of our algorithm and that it runs in polynomial time. The proof is provided in the extended version of the paper [50].

Theorem 5.1 (Soundness)**.** *If the algorithm outputs a value t, then t is a threshold for the DiffCost problem. Furthermore, the algorithm runs in polynomial time for the fixed values of parameters d and K.*

Our method computes polynomial PFs and anti-PFs and hence the computed cost bounds are also polynomials over program variables. However, it is known that computing tight cost bounds in programs with more complex control-flow may require disjunctive expressions involving piecewise-linear operators such as max or min [10, 25, 46]. Hence, for such programs, it might not be possible to compute polynomial cost bounds that provide desired precision guarantees.

Computing disjunctive cost bounds has been the focus of several works on single program cost analysis. In particular, the work of Carbonneaux et al. [10] also employs a constraint solving-based approach to compute *piecewise linear upper bounds* on cost usage in programs. An interesting direction for future work would be to extend the technique to polynomial and lower bounds on cost usage in order to combine it with our framework for differential cost analysis.

Proving symbolic polynomial bounds. We conclude this section by describing how our algorithm may be adapted to proving symbolic polynomial bounds on the difference in cost usage. Let p be a polynomial function over the set of program variables V and suppose that we want to prove the differential cost assertion

$$\forall \mathbf{x} \in \Theta_0.\ CostSup_{\mathcal{T}^{\text{new}}}(\ell_0^{\text{new}}, \mathbf{x}) - CostInf_{\mathcal{T}^{\text{old}}}(\ell_0^{\text{old}}, \mathbf{x}) \leq p(\mathbf{x}).$$

In order to use our algorithm to prove such an assertion, one should drop the minimization objective in Step 4 and replace every appearance of the constant threshold t in the algorithm description by the polynomial p. By doing this and by requiring that the maximal polynomial degree parameter d is greater than or equal to the degree of p, we may use our algorithm to prove the desired differential cost assertion. The only difference in using our algorithm to prove a polynomial bound p as opposed to a concrete threshold value t would occur in Step 2, as this change would introduce polynomial p in the differential cost constraint. However, as the polynomial would appear on the right-hand-side of the implication, the conversion in Step 3 of the algorithm would still reduce the DiffCost problem to an LP instance and the rest of the algorithm would proceed analogously as in the case of the concrete threshold value t.

6 Empirical Evaluation

In this section, we evaluate our approach to differential cost analysis by considering the following research question:

Tightness of Differential Thresholds. Does the simultaneous derivation of potential and anti-potential functions with a threshold value t yield tight differential bounds?

To address this question, we consider 19 program pairs as benchmarks on which we perform differential cost analysis in Table 1. In the end, our method computed tight thresholds for 74% (14/19) of the benchmarks. There were 2 benchmarks where we failed to compute any threshold, and we discuss the reason behind the failure below. The run times of our tool (including invariant generation, extraction of constraints, and linear programming) are of the order of a few seconds and suggest that run time is not the bottleneck.

To evaluate the tightness of the differential thresholds we can derive with simultaneous computation of potentials and anti-potentials, we consider two classes of benchmarks. The first consist of examples that increase the cost (i.e., the tightest possible threshold is non-zero), while the second consist of examples that do not change the cost but have non-trivial cost-preserving changes (e.g., would be non-trivial to align for relational reasoning).

Table 1. Tightness of differential thresholds. For each benchmark, we note the *Tight* differential threshold bound (i.e., the maximal difference in cost usage that can be attained, which is determined manually) and the one we *Computed*. Computed thresholds that are tight and match the maximal difference are given in **bold**. Finally, the *Time* column shows the time (in seconds) taken by our tool to compute the threshold. The ✗ indicates the cases where we were unable to compute a threshold. The programs are drawn the literature on cost analysis and semantic differencing, with the first group of 10 from Gulwani et al. [23], the next 5 from Gulwani and Zuleger [25], and the last 4 from Partush and Yahav [40, 41]. The program names are the original names used in the works from which the examples are taken, and consist of methods that are between 10 and 20 lines of code. The * marks cases where the off-the-shelf invariant generators miss some simple, expected invariants about the loop bounds being satisfied upon entering the loop bodies and so where we slightly strengthened them.

Benchmark	Threshold (n)		Time (s)
	Tight	Computed	
Non-Zero Tight Threshold			
Dis1	**100**	**100**	2.5
Dis2	**100**	**100**	3.4
NestedMultiple	**100**	**100**	4.3
NestedMultipleDep*	**9900**	**9900**	1.5
NestedSingle	**101**	**101**	0.9
SequentialSingle	**100**	**100**	0.9
SimpleMultiple	**100**	**100**	1.5
SimpleMultipleDep	10000	10100	1.4
SimpleSingle	**100**	**100**	0.7
SimpleSingle2	100	197	2.0
Ex2	**99**	**99.94**	2.3
Ex4	**201**	**201**	0.9
Ex5	100	✗	2.1
Ex6	**99**	**99.01**	1.4
Ex7	1	✗	1.4
Zero Tight Threshold			
ddec	0	73896.43	0.9
ddec modified	**0**	**0**	0.9
nested*	**0**	**0**	2.0
sum	**0**	**0.5**	1.7

In the first class of benchmarks, we obtain programs from the single-program cost analysis literature (specifically, [23, 25]) to emphasize testing the cost reasoning capability. These programs involve combinations of single or nested loops with conditional branching and non-determinism, and present complex looping patterns. We consider these as representatives of the class of real world examples that involve, for example, iterating through collections with branching or non-deterministic behavior of a function called through an interface. These are code patterns that can often lead to unpredictable resource usage and for which automated differential cost analysis is particularly beneficial. From Gulwani and Zuleger [25], we omit Ex1 and Ex3, as the off-the-shelf tool we use to translate C programs to transition systems (C2fsm [20]) does not support booleans and pointers.

From each of these programs with interesting resource usage, we produce a revision or program pair for differential cost analysis as follows. For the first program, we make it incur a cost of 1 for each loop iteration so that the total cost usage corresponds to the loop bound for the program. This choice is to match the original intent of the these benchmarks, as they are drawn from work on loop-bound analysis. In the second program, we pick either a nested loop or an if-branch to incur a cost of 1. This choice of selecting a nested loop or an if-branch to incur cost makes the differential cost analysis non-trivial (i.e., the second program does not have the same cost as the first nor 0 cost, and differential cost analysis requires reasoning about different program behavior in each loop iteration).

In the second class of benchmarks, we obtain examples from the semantic differencing literature (specifically, [40, 41]) to emphasize testing the differential reasoning capability. These are examples of semantically equivalent program pairs, so their relative cost is 0 on every input. In 2 out of the 3 examples, there is a syntactic difference that does not allow program alignment—making it difficult to apply a relational approach. The remaining program pairs from that work either involve boolean variables and pointers, or are straightline programs for which cost analysis is trivial.

For all of the examples, our method computes polynomial cost bounds with the maximal polynomial degree and the maximal number of terms in products in Step 3 of our algorithm being $d = K = 2$, except for 'nested' in which the total cost usage is cubic so we use $d = K = 3$. Finally, for each uninitialized program variable we assume that its initial value is in the interval $[1, 100]$. In cases when a loop bound is defined by the difference between the variable value and its symbolic initial value, we assume that this difference is in the interval $[1, 100]$. We do this to bound initial variable values, so that the difference in cost usage between programs is also bounded and we may use the aforementioned program pairs for a meaningful evaluation of our approach. In Dis2, we in addition assume an initial ordering of variable values to avoid the need for disjunctive reasoning.

Limitations and Threats to Validity. We inspected examples for which our approach is either not able to compute a bound or for which the computed bound is not tight. In particular, we observe that SimpleMultipleDep, SimpleSingle2, Ex5, Ex7 and ddec all require disjunctive reasoning in order to compute tight cost bounds. Disjunctive reasoning is somewhat

orthogonal, but it does indicate that disjunctive reasoning is a limitation as-is (as discussed in Section 5) and is indeed important for differential cost analysis like other analysis domains. The program 'ddec modified' is a modification of ddec that does not require disjunctive reasoning for cost analysis, and we see that for this example our method computes a tight bound on cost difference.

Another interesting thing to note in Table 1 is the small imprecision in bounds computed for Ex2, Ex4 and 'sum'. Observe that the computed bounds for these examples slightly exceed the maximal difference in cost usage, with the difference being smaller than 1 (as a result of real-valued linear programming). Nevertheless, since we consider programs with integer variables and costs, the computed bound is tight.

A natural threat to validity is whether these benchmarks are representative. We have attempted to mitigate this threat by considering two classes of benchmarks focusing on different aspects and considering looping patterns for common code like iterating over collections. There are few works that study differential cost analysis, and [16, 42] experimentally evaluate their approaches on functional programs that manipulate list-like objects. Since we consider differential cost analysis in general imperative programs for which, to the best of our knowledge, there is no existing benchmark set, we created our own benchmark set by collecting example imperative programs from the literature.

Implementation. In order to empirically evaluate our approach, we implemented a prototype tool which takes C programs as input and uses C2fsm [20] to translate them to equivalent transition systems. We note that C2fsm supports a slightly out-of-date dialect of C that does not allow boolean data types or constructs such as structs or pointers. However, it supports general numerical data types and control-flow constructs, making it is sufficient for our evaluation. For invariant generation we use Aspic [20] and Sting [44], and we use Gurobi [26] to solve linear programs. All experiments were run on Ubuntu with an Intel(R) Core(TM) i5-8250U CPU at 1.60GHz and with 16GB of RAM.

7 Precision Guarantees on Bounds for a Single Program

As noted in Section 2, while the motivation for simultaneously computing potential and anti-potential functions is to address the differential cost analysis problem, an additional consequence of our approach is that it suggests a way to compute cost bounds with *precison guarantees*. We now show how our approach can be adapted to compute upper and lower bounds on cost incurred in a single program with guarantees on the precision of computed bounds. And we have not seen other cost analyses that provide such guarantees on the quality of computed cost bounds.

Let $\mathcal{T} = (L, V, \to, \ell_0, \Theta_0)$ be a terminating transition system that models a program whose cost usage we wish to analyze. We may compute bounds on the program's cost usage by naturally adapting our algorithm presented in Section 5 to compute (1) a PF ϕ in $\mathcal{T}$, (2) an anti-PF χ in $\mathcal{T}$, and (3) a value p that satisfies

$$\forall \mathbf{x} \in \Theta_0.\, \phi(\ell_0, \mathbf{x}) - \chi(\ell_0, \mathbf{x}) \leq p.$$

The resulting algorithm simultaneously computes ϕ, χ and p by reduction to an LP instance that minimizes p. The computed value of p is a *bound on precision* of both the computed upper bound ϕ and the lower bound χ on cost incurred in $\mathcal{T}$. The following theorem proves that our algorithm is sound and that it indeed provides precision guarantees. The proof can be found in the extended version of the paper [50].

Theorem 7.1. *Let $\mathcal{T}$ be a terminating transition systems. Suppose that ϕ is a PF and that χ is an anti-PF in $\mathcal{T}$. Then, for each initial variable valuation $\mathbf{x} \in \Theta_0$, we have that*

$$\mathit{CostSup}_{\mathcal{T}}(\ell_0, \mathbf{x}) - \mathit{CostInf}_{\mathcal{T}}(\ell_0, \mathbf{x}) \leq \phi(\ell_0, \mathbf{x}) - \chi(\ell_0, \mathbf{x}).$$

In particular, if p satisfies $\phi(\ell_0, \mathbf{x}) - \chi(\ell_0, \mathbf{x}) \leq p$ for each $\mathbf{x} \in \Theta_0$, then for any run ρ that starts in some initial state $(\ell_0, \mathbf{x})$ with $\mathbf{x} \in \Theta_0$ we have $0 \leq \mathit{Cost}_{\mathcal{T}}(\rho) - \chi(\ell_0, \mathbf{x}) \leq p$ and $0 \leq \phi(\ell_0, \mathbf{x}) - \mathit{Cost}_{\mathcal{T}}(\rho) \leq p$. Hence, p is a bound on the precision of the upper cost bound defined by ϕ and the lower cost bound defined by χ in $\mathcal{T}$.

We note that the lack of disjunctive reasoning in our method could result in weaker bounds for programs in which disjunctive reasoning for computing tight cost bounds is needed, when compared to some approaches that do compute them [10, 25, 46]. However, as new techniques are developed that compute tighter bounds, our approach may offer a way to get both tight bounds and precision guarantees.

8 Related Work

Differential cost analysis. The existing works on differential cost analysis consider functional programs and propose relational type and effect systems to reason about relative cost between programs [14, 16, 43], and the relational type and effect system of [42] allows reasoning about functional programs with mutable arrays. The relational analysis is done by syntactically aligning two programs and considering relational types that capture the difference in cost incurred in two programs. Once the alignment is no longer possible, these works use unary types to allow cost analysis in a single program. As discussed in Section 1, our approach offers an alternative with different trade-offs. For example, when given a pair of syntactically similar programs, relational type systems naturally exploit this similarity through syntactic alignment of programs, which may simplify differential cost analysis [14]. And in contrast to the approaches based on relational type systems, our method is not compositional. It reduces differential cost analysis to solving a linear program, which is a global optimization problem whose constraints

may depend on all parts of the program. An interesting direction of future work is to extend our method to generate method summaries [24] that can express quantitative specifications on cost (to make it more compositional).

The Infer Static Analyzer [19] performs a kind of differential cost analysis on industrial size codebases using a worst-case execution time (WCET) analysis [9]. It considers C or Java programs and performs unary cost analysis on each program to compute two polynomial cost upper bounds, which are then compared. A warning is raised if there is a jump in polynomial degree of incurred cost [15]. Thus, their method detects increases in resource usage only when there is an increase in polynomial degree. However, it is impressively scalable and in many cases successfully detects increases in cost polynomial degrees. Given that Infer also focuses on imperative programming languages, it may be possible to couple it with our approach in order to obtain a tool which scales to large codebases, but which allows a sound and precise differential cost analysis for parts of programs that are deemed performance critical. One possible strategy is to use our approach for precise amortized reasoning locally within a global worst-case reasoning framework [37].

Static (unary) cost analysis. Static cost analysis for single programs is a classical and well studied problem. There are many existing methods, and most works focus on computing upper bounds on cost usage with techniques based on amortized analysis [10, 28–31], type systems [3, 32, 34, 35], term-rewriting and abstract interpretation [8, 22, 23, 25], ranking functions [1], invariant generation [33] or the analysis of abstract program models [45, 46, 49]. In particular, [10, 28–31] also compute potential functions for amortized analysis by reduction to linear programming. The approach of Kincaid et al. [33] that is based on invariant generation can be used to obtain both upper and lower bounds on cost usage with bounds that may involve piecewise-linear operators max and min. Thus, similarly to Carbonneaux et al. [10], it would be interesting to consider the possibility of combining it with our framework towards obtaining piecewise-polynomial bounds for differential cost analysis. Computing lower bounds on program run time has been considered [21]. Ngo et al. [39] proposes a type system for verifying that a functional program has constant resource usage, which has important implications in security and prevents leakage of confidential information under side-channel attacks that exploit non-constant run times or energy consumption. Their type system defines a potential-like function whose values before and after evaluating an expression differ exactly by the cost of the evaluation. This corresponds to strengthening the *sufficiency* preservation condition of PFs and the *insufficiency* preservation condition of anti-PFs by imposing the strict equality "=" sign. By relaxing type judgements with "≥" or "≤" signs, the type system is relaxed to a type system for computing upper or lower bounds on cost usage and type checking would correspond to computing a PF or an anti-PF, respectively. Computing upper and lower bounds on the expected cost usage in probabilistic programs has also been considered [4, 38, 48]. In particular, our technique for translating the defining conditions of PFs and anti-PFs into linear constraints has similarities to that of Wang et al. [48], which defines a probabilistic variant of potential functions called upper and lower cost supermartingales and uses Handelman's theorem for their computation. Handelman's theorem and other results from algebraic geometry are used by Chatterjee et al. [11] for computing probability 1 termination certificates and bounds on termination time. While Ngo et al. [38] and Wang et al. [48] consider computation of both upper and lower cost bounds, they study unary cost analysis. If one attempted to directly adapt these methods to differential cost analysis by computing two cost bounds and comparing them, this would be precisely the naïve approach discussed in Section 1 that is problematic for precision. This limitation of the naïve approach was also pointed out in prior work on differential cost analysis and was used to motivate the use of relational type systems [14]. In contrast, we compute upper and lower cost bounds for a pair of programs *together with a threshold value* to address the differential cost analysis problem; it is this simultaneous computation that side-steps the limitations of the naïve approach.

Constraint solving-based program analysis. Constraint solving-based techniques are a classical approach to program analysis [24], that have been used for multiple static analyses including the synthesis of ranking functions for termination analysis [1, 7, 18], proving non-termination [13, 36], invariant generation [12, 17], reachability [2], as well as several methods for cost analysis that we discussed above.

9 Conclusion

We present a novel approach to differential cost analysis for imperative programs that uses potential and anti-potential functions to reason about the difference in incurred cost. A threshold value on the maximal difference in cost between two program versions is computed simultaneously with a potential function that provides an upper bound on the cost incurred in the new version and an anti-potential function that provides a lower bound on the cost incurred in the old one. This dual potential-based method side steps the need for and limitations of version alignment and offers a complimentary approach to ones based on relational reasoning. To automatically derive the potential function, the anti-potential function, and the threshold for differential cost analysis, we employ a constraint solving-based approach that has the benefit of using off-the-shelf invariant generators and linear program solvers, as well as supporting optimization of the threshold bound.

Acknowledgements

We thank Shaun Willows, Thomas Lugnet, and the Living Room Application Vending team for suggesting threshold bounds as a developer-friendly way to interact with a differential cost analyzer, and we thank Jim Christy, Daniel Schoepe, and the Prime Video Automated Reasoning team for their support and helpful suggestions throughout the project. We also thank Michael Emmi for feedback on an earlier version of this paper. And finally, we thank the anonymous reviewers for their useful feedback and Aws Albarghouthi for shepherding the final version of the paper. Đorđe Žikelić was also partially supported by ERC CoG 863818 (FoRM-SMArt).

References

[1] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6337)*, Radhia Cousot and Matthieu Martel (Eds.). Springer, 117–133. https://doi.org/10.1007/978-3-642-15769-1_8

[2] Ali Asadi, Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Mohammad Mahdavi. 2021. Polynomial reachability witnesses via Stellensätze. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 772–787. https://doi.org/10.1145/3453483.3454076

[3] Martin Avanzini and Ugo Dal Lago. 2017. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.* 1, ICFP (2017), 43:1–43:29. https://doi.org/10.1145/3110287

[4] Martin Avanzini, Georg Moser, and Michael Schaper. 2020. A modular cost analysis for probabilistic programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 172:1–172:30. https://doi.org/10.1145/3428240

[5] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Math. Struct. Comput. Sci.* 21, 6 (2011), 1207–1252. https://doi.org/10.1017/S0960129511000193

[6] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 14–25. https://doi.org/10.1145/964001.964003

[7] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear Ranking with Reachability. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3576)*, Kousha Etessami and Sriram K. Rajamani (Eds.). Springer, 491–504. https://doi.org/10.1007/11513988_48

[8] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2016. Analyzing Runtime and Size Complexity of Integer Programs. *ACM Trans. Program. Lang. Syst.* 38, 4 (2016), 13:1–13:50. http://dl.acm.org/citation.cfm?id=2866575

[9] Stefan Bygde. 2010. Static WCET Analysis based on Abstract Interpretation and Counting of Elements. http://www.es.mdh.se/publications/1789-

[10] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 467–478. https://doi.org/10.1145/2737924.2737955

[11] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 3–22. https://doi.org/10.1007/978-3-319-41528-4_1

[12] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2020. Polynomial invariant generation for non-deterministic recursive programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 672–687. https://doi.org/10.1145/3385412.3385969

[13] Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, and Đorđe Žikelić. 2021. Proving non-termination by program reversal. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1033–1048. https://doi.org/10.1145/3453483.3454093

[14] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 316–329. https://doi.org/10.1145/3009837.3009858

[15] Ezgi Çiçek, Mehdi Bouaziz, Sungkeun Cho, and Dino Distefano. 2020. Static Resource Analysis at Scale (Extended Abstract). In *Static Analysis - 27th International Symposium, SAS 2020, Virtual Event, November 18-20, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12389)*, David Pichardie and Mihaela Sighireanu (Eds.). Springer, 3–6. https://doi.org/10.1007/978-3-030-65474-0_1

[16] Ezgi Çiçek, Weihao Qu, Gilles Barthe, Marco Gaboardi, and Deepak Garg. 2019. Bidirectional type checking for relational properties. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 533–547. https://doi.org/10.1145/3314221.3314603

[17] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2725)*, Warren A. Hunt Jr. and Fabio Somenzi (Eds.). Springer, 420–432. https://doi.org/10.1007/978-3-540-45069-6_39

[18] Michael Colón and Henny Sipma. 2001. Synthesis of Linear Ranking Functions. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2031)*, Tiziana Margaria and Wang Yi (Eds.). Springer, 67–81. https://doi.org/10.1007/3-540-45319-9_6

[19] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. https://doi.org/10.1145/3338112

[20] Paul Feautrier and Laure Gonnord. 2010. Accelerated Invariant Generation for C Programs with Aspic and C2fsm. *Electron. Notes Theor. Comput. Sci.* 267, 2 (2010), 3–13. https://doi.org/10.1016/j.entcs.2010.09.014

[21] Florian Frohn, Matthias Naaf, Marc Brockschmidt, and Jürgen Giesl. 2020. Inferring Lower Runtime Bounds for Integer Programs. *ACM Trans. Program. Lang. Syst.* 42, 3, Article 13 (Oct. 2020), 50 pages. https://doi.org/10.1145/3410331

[22] Sumit Gulwani. 2009. SPEED: Symbolic Complexity Bound Analysis. In *Computer Aided Verification, 21st International Conference, CAV 2009,*

Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643), Ahmed Bouajjani and Oded Maler (Eds.). Springer, 51–62. https://doi.org/10.1007/978-3-642-02658-4_7

[23] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 127–139. https://doi.org/10.1145/1480881.1480898

[24] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2008. Program analysis as constraint solving. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 281–292. https://doi.org/10.1145/1375581.1375616

[25] Sumit Gulwani and Florian Zuleger. 2010. The reachability-bound problem. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 292–304. https://doi.org/10.1145/1806596.1806630

[26] Gurobi Optimization, LLC. 2021. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[27] David Handelman. 1988. Representing polynomials by positive linear functions on compact convex polyhedra. *Pacific J. Math.* 132, 1 (1988), 35–62.

[28] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate amortized resource analysis. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 357–370. https://doi.org/10.1145/1926385.1926427

[29] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 781–786. https://doi.org/10.1007/978-3-642-31424-7_64

[30] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 359–373. https://doi.org/10.1145/3009837.3009842

[31] Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polynomial Potential. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6012)*, Andrew D. Gordon (Ed.). Springer, 287–306. https://doi.org/10.1007/978-3-642-11957-6_16

[32] Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 185–197. https://doi.org/10.1145/604131.604148

[33] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas W. Reps. 2017. Compositional recurrence analysis revisited. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 248–262. https://doi.org/10.1145/3062341.3062373

[34] Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*. IEEE Computer Society, 133–142. https://doi.org/10.1109/LICS.2011.22

[35] Ugo Dal Lago and Barbara Petit. 2013. The geometry of types. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 167–178. https://doi.org/10.1145/2429069.2429090

[36] Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2014. Proving Non-termination Using Max-SMT. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 779–796. https://doi.org/10.1007/978-3-319-08867-9_52

[37] Tianhan Lu, Bor-Yuh Evan Chang, and Ashutosh Trivedi. 2021. Selectively-Amortized Resource Bounding. In *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12913)*, Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi (Eds.). Springer, 286–307. https://doi.org/10.1007/978-3-030-88806-0_14

[38] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 496–512. https://doi.org/10.1145/3192366.3192394

[39] Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 710–728. https://doi.org/10.1109/SP.2017.53

[40] Nimrod Partush and Eran Yahav. 2013. Abstract Semantic Differencing for Numerical Programs. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7935)*, Francesco Logozzo and Manuel Fähndrich (Eds.). Springer, 238–258. https://doi.org/10.1007/978-3-642-38856-9_14

[41] Nimrod Partush and Eran Yahav. 2014. Abstract semantic differencing via speculative correlation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 811–828. https://doi.org/10.1145/2660193.2660245

[42] Weihao Qu, Marco Gaboardi, and Deepak Garg. 2019. Relational cost analysis for functional-imperative programs. *Proc. ACM Program. Lang.* 3, ICFP (2019), 92:1–92:29. https://doi.org/10.1145/3341696

[43] Ivan Radicek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2018. Monadic refinements for relational cost analysis. *Proc. ACM Program. Lang.* 2, POPL (2018), 36:1–36:32. https://doi.org/10.1145/3158124

[44] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2004. Constraint-Based Linear-Relations Analysis. In *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3148)*, Roberto Giacobazzi (Ed.). Springer, 53–68. https://doi.org/10.1007/978-3-540-27864-1_7

[45] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 745–761. https://doi.org/10.1007/978-3-319-08867-9_50

[46] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2017. Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints. *J. Autom. Reason.* 59, 1 (2017), 3–45. https://doi.org/10.1007/s10817-016-9402-4

[47] Robert Endre Tarjan. 1985. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods* 6, 2 (1985), 306–318.

[48] Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi. 2019. Cost analysis of nondeterministic probabilistic programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 204–220. https://doi.org/10.1145/3314221.3314581

[49] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. 2011. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6887)*, Eran Yahav (Ed.). Springer, 280–297. https://doi.org/10.1007/978-3-642-23702-7_22

[50] Đorđe Žikelić, Bor-Yuh Evan Chang, Pauline Bolignano, and Franco Raimondi. 2022. Differential Cost Analysis with Simultaneous Potentials and Anti-potentials. https://doi.org/10.48550/ARXIV.2204.00870